Optimising Java RMI Programs by Communication Restructuring

Kwok Cheung Yeung and Paul H. J. Kelly

Department of Computing
Imperial College, London, UK

Abstract. We present an automated run-time optimisation framework that can improve the performance of distributed applications written using Java RMI whilst preserving its semantics.
Java classes are modified at load-time in order to intercept RMI calls as they occur. RMI calls are not executed immediately, but are delayed for as long as possible. When a dependence forces execution of the delayed calls, the aggregated calls are sent over to the remote server to be executed in one step. This reduces network overhead and the quantity of data sent, since data can be shared between calls. The sequence of calls may be cached on the server side along with any known constants in order to speed up future calls. A remote server may also make RMI calls to another remote server on behalf of the client if necessary.
Our results show that the techniques can speed up distributed programs significantly, especially when operating across slower networks. We also discuss some of the challenges involved in maintaining program semantics, and show how the approach can be used for more ambitious optimisations in the future.

1 Introduction

Frameworks for distributed programming such as the Common Object Resource Broker Architecture (CORBA) [9] and Java Remote Method Invocation (RMI) [12] aim to provide a location-transparent object-oriented programming model, but do not completely succeed since the cost of a remote call may be several orders of magnitude greater than a local call due to marshalling overheads and relatively slow network connections. This means that developers must explicitly code with performance in mind, leading to reduced productivity and increased program complexity.

The usual approach to optimising distributed programs in general has been to optimise the connection between the communicating hosts, fine-tuning the remote call mechanism and the underlying communication protocol to cut the overhead for each call to a minimum. Although this leads to a general speed-up, it does not help the performance of programs that are slow due to their structure (e.g. using many fine-grained methods instead of a few coarse-grained methods). Our approach towards solving this problem has been to consider all communicating nodes as part of one large program, rather than many disjoint ones.

We delay the execution of remote calls on the client for as long as possible until a dependency on the delayed calls blocks further progress. At this point, the delayed calls are executed in one step, after which the blocked operation may proceed. By delaying

the execution of remote calls, we build up a knowledge of the context in which calls were made on the client. This enables us to find opportunities for optimisations between calls that would have been lost had the calls been executed immediately.

1.1 Contributions

- We present an optimisation tool which can improve performance of Java/RMI applications by combining static analysis of application bytecode with run-time optimisation of sequences of remote operations. This tool operates on unmodified Java RMI applications, and runs on a standard JVM.
- By aggregating sequences of remote calls to the same server, the total number of message exchanges is reduced. By avoiding redundant parameter and result transfers, total amount of data transferred can also be reduced. When calls to different servers are aggregated together, results can be forwarded directly from one server to another, bypassing the client in some cases.
- We show how run-time overheads can be reduced by caching execution plans at the servers.
- We demonstrate the use of the tool using a number of examples.

The framework presented here provides the basis for a programme of research aimed at extending aggressive optimisation techniques across distributed systems, and deploying the results in large-scale industrial systems. We conclude with a discussion of the potential for the work, and the challenges that remain.

1.2 Structure

We begin in Section 2 with a discussion of related work. We then cover the runtime optimisation framework used to implement our optimisations at a high-level in Section 3. We proceed to cover the optimisations performed in Section 4, and the challenges involved in maintaining the semantics of the original application in Section 5. We then present some performance results in Section 6 and finish off with some suggestions for future work in Section 7 and conclude in Section 8.

2 Related Work

Most work on optimising RMI has concentrated on reducing the run-time overhead of each remote call by reducing the amount of work done per-call or by using more lightweight network protocols. Examples include the UKA serialisation work [14], KaRMI [13], and R-UDP [10]. Similar work has been done on CORBA by Gokhale and Schmidt [7].

Asynchronous RPC [11, 15] aims to overlap client computation with communication and remote execution, replacing results with ‘promises’, which block the client only when actually used.

A more ambitious approach is the concept of caching the state of a remote-object locally [10]. This works well provided that most operations on cached objects are reads.

However, a write operation incurs high penalties for all users of the cached object, since the client has to wait for invalidation of all copies of the object to finish before proceeding. The first request for invalidated data will also incur an extra delay as the server fetches it from the client that performed the last update.

A later implementation of remote-object caching [5] implements the notion of *reduced objects* where only a subset of the remote-object state is cached on the server. The subset that is cached depends on the properties of the invoked methods — e.g. if a called method only accesses immutable variables, then those variables can be cached on the client without needing to deal with consistency issues.

Neither of these approaches to RMI optimisation conflict with our aggregation optimisations, and although we have not done so ourselves, these optimisations could theoretically be combined. It may be argued that our optimisations are made redundant under certain circumstances (e.g. if the aggregated calls are cached locally).

The concept of aggregating numerous small operations into a single larger operation is very old, and appears in numerous other contexts, especially in the hardware domain. In the context of RPC mechanisms, concepts such as stored procedures in database systems or commands in IBM's San Francisco [3] project are also capable of aggregating calls, but these are explicit mechanisms. Implicit call aggregation is much rarer and harder to implement. One example would be the concept of batched futures [2] in the Thor database system.

3 The Veneer Framework

The RMI optimisations are based on top of Veneer, which is a generalised framework that we have developed for the purpose of easing the development of run-time optimisation techniques. This framework is written in standard Java, using the BCEL [4] library for bytecode generation and the Soot [16] library for program analysis. Veneer is not tied to any particular JVM implementation, which is essential since it is likely be used in a heterogeneous environment. We refer to Veneer as a 'virtual JVM', since it behaves like a highly configurable Java virtual machine, without actually being one.

The framework presents a simplified model of the Java run-time environment, working with what appears to be a simple interpreter, called an *executor*. A basic executor is shown in Figure 1, which executes a method with no modifications whatsoever.

When a method that we are interested in is called, control passes to our executor instead of the original method. The executor is initialised with an *execution plan*, which is essentially a control-flow graph of the method, with executable code-blocks forming the nodes. The executor sits in a loop which executes the current block, then sets the current block to the next block in line to be executed.

The power of this framework lies in the fact that the plan is a first-order object that we can change while the executor is still running, effectively modifying the code that will be executed. The executor has full control over the process of method execution between blocks, such that we can perform operations such as jumping to arbitrary code-blocks, modifying local variables or timing operations if necessary.

We minimise the interpretive overhead by delegating as much work as possible to the underlying JVM, and by making the code-blocks as coarse as possible. There is also

an option to permit blocks to run continuously without returning to the executor, though certain block types will always force a return.

The mapping of byte-code to code-blocks in the plan and the methods affected by our framework are determined by a plug-in policy class. The policy class also contains numerous call-back methods that are invoked on certain events, such as the initial loading of a class.

```
public class BasicExecutor extends Executor{
  public int execute() throws Exception {
    while (block != null
           && !lockWasReleased()) {
      int next = -1;

      try {
        next = block.execute(this);
        block = plan.getBlock(next);
      } catch (ExecuteException e) {
        // Pass control to exception handler
        block = plan.getExceptionHandler(e);

        // Propagate exception if no handler
        if (block == null)
          throw e.getException();

        locals[1] = e.getException();
      }
    }

    return next;
  }
}
```

Fig. 1. Structure of a basic executor

4 Optimisations

In this section we detail the RMI optimisations that have been implemented. The examples used to illustrate the optimisations are deliberately simplified for clarity.

4.1 Call Aggregation

Delaying calls to form call aggregates is the core technique upon which this project is based. It is an important optimisation in its own right, and furthermore can also open up further optimisation opportunities. For example, consider the following code fragment:

```
void m(RemoteObject r, int a) {
    int x = r.f(a);
    int y = r.g(x);
    int z = r.h(y);

    System.out.println(z);
}
```

This program fragment incurs three remote method calls, with six data transfers. However, for this example, we can do better:

- Since all three calls are to the same remote object, they can be aggregated into a single large call, such that the number of times that call overhead is incurred is reduced to one (see Figure 2).
- *x* is returned as the result of the call to **f** from the remote server, but is subsequently passed back to it during the next call. The same occurs with the variable *y*. If the values of *x* and *y* were retained by the remote object between remote method calls, then the number of communications could be reduced from six to four.
- The variables *x* and *y* are unused by the client except as arguments to remote calls on the remote object from which they originated. *x* and *y* may therefore be considered as dead variables from the client's point of view, and there is no need for their value to be passed back to the client at all, thereby further reducing the total number of remote transactions down to just two messages with payloads of size *int*.

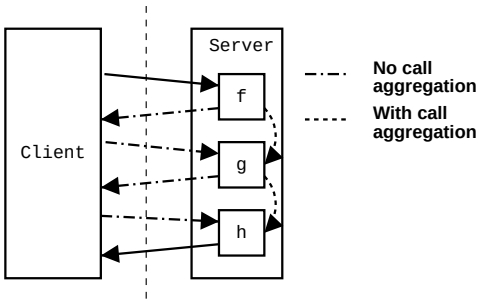


Fig. 2. Example of call aggregation

Client-side Implementation We have created a Veneer policy that only affects methods that are statically determined to contain potentially remote method calls. Calls are deemed to be potentially remote if they are invoked via an interface, and have `java.rmi.RemoteException` or one of its super-classes on the throw list. A run-time check is later used to ensure that the potential remote call is actually remote. Note that it is not sufficient just to check that the receiver of the call implements

`java.rmi.Remote` since the object could be invoked directly instead of via RMI, and some remote calls may be missed if we are supplied with a non-remote interface that is actually a remote object that implements a remote child of our interface.

The client runs under the control of the Veneer framework using this policy. If the executor encounters a confirmed remote call during the course of execution, then it places the call within a queue and proceeds to the next instruction. Sequences of adjacent calls to the same remote object are grouped together into *remote plans*. Remote plans also contain metadata regarding the calls, such as variable liveness and data dependencies. Calls to other remote objects will not force execution unless the target of the call is defined by a previous delayed call, leading to a control dependency. However, even this condition is relaxed by server forwarding, detailed in Section 4.2.

When a non-remote block is encountered with delayed calls remaining in the queue, a decision has to be made whether or not to force execution of the calls. In general, it is safe to execute the current block without forcing if there are no dependencies between the current instruction and the delayed operations. If dependencies exist or if it is impossible to tell, then we must force execution.

We detect data dependencies by noting attempts to access data returned by RMI calls. Since the results of RMI calls are constructed by deserialising the data returned by the server, there can be no other references to the returned data except for the local that the result of the remote call was placed in. We therefore regard local code that accesses locals that should contain the results of RMI calls as being dependent on the delayed calls.

This scheme is rather conservative, such that even simple assignments from one local variable to another can force the execution of the delayed plans. We hope to improve this in the future using improved static analysis. Also, it cannot detect indirect data dependencies — for example, if the RMI call modifies a remote database which the client proceeds to access using another API, then that access will go unnoticed.

When executing local code in the presence of delayed remote calls, we must ensure that the variables used by the delayed calls are not overwritten or modified by the local code. This is done by making a copy of all locals supplied to the delayed calls that may be touched by the local code.

On forcing execution, the queue of delayed remote plans is traversed, with plans being sent one-by-one, along with the set of data used by the plan, to the corresponding *remote proxy* on the server-side via standard RMI invocation to be executed. The proxy call may either return successfully or throw an exception.

If the call returns successfully, then the variables defined by the plan that are still live are copied back into the locals set of the executing method. If an exception was thrown, then the executor goes through the normal process of finding a handler for the exception within the method, and propagating it up the call chain if one is not found.

The same Veneer policy also runs a remote proxy server on startup, which first registers itself in a naming service via JNDI. The proxy keeps track of all remote objects present on the JVM by inserting a small callback into the constructors of all remote classes at load time[1].

[1] This may lead to a potential security hole since this may occur before the remote object has been exported for remote access

Clients obtain handles to proxies using the standard naming services via JNDI. When a client first encounters a new remote stub, it broadcasts it to all known proxies. The proxy that handles the remote object denoted by the stub will identify itself. Remote plans containing calls on that stub will subsequently be sent to the identified proxy. The stub-to-proxy mapping is cached on the client for speed.

Remote plans sent to the proxy are executed by an executor, which simply executes the calls one-by-one. The calls are made directly on the remote object rather than via another RMI invocation. However, care must be taken due to the semantic differences between local and RMI calls (see Section 5.1).

When finished, the proxy only sends the variables that are live in the client program at the point where execution was forced. The live set is calculated using the metadata supplied with the remote-calls.

4.2 Server Forwarding

Server forwarding takes advantage of the fact that servers typically reside on fast connections, whilst the client-server connection can often be orders of magnitude slower. Consider this sequence of calls:

```
x = r1.f();
y = r2.f();
z = r3.f(x,y);
```

The first two methods invoked on *r1* and *r2* are returning objects that are subsequently used as arguments to a method on another remote object *r3*. In this situation, the client is acting as a router for messages between *r1*, *r2* and *r3*. It would be better for *r1* and *r2* to communicate with *r3* directly, such that no constraints are set as to which path is taken between the two servers. Also, if *x* or *y* are dead, then they need not be returned to the client.

Forwarding is also necessary for efficient aggregation of factory patterns. e.g.

```
a = r.newObject();
b = a.f();
```

Without forwarding in place, we would need to force after the call to `newObject` because `a` is used as the receiver for the next remote call — without knowing the value of `a`, we would not where to send the remote plan, or what object to invoke `f` on.

Implementation Server forwarding is implemented on top of call aggregation in a preprocessing step just before execution on the remote proxies, by grouping remote plans on differing remote objects together. When a remote proxy encounters a plan that is handled by another remote proxy, it will forward the plan onto that proxy automatically.

At present, the remote plans are composed of straight-line sequences of remote calls to the same object bundled together. We will refer to these units as *call clusters*. We use the following heuristics to decide when to group clusters with differing destinations:

– Plans that are delivered to the same remote proxy should be grouped together

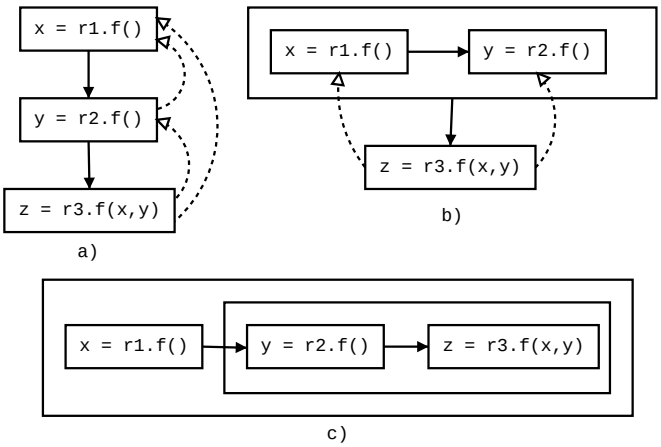


Fig. 3. Implementation of call forwarding: a) Arcs are placed between the calls to r1–r3 and r2–r3 (due to data dependence) and r1–r2 (due to co-location), b) Current cluster is the call to r1 — we append the call to r2 due to the r1–r2 arc, c) Current cluster is the call to r2 — we prepend the call to r3 due to the r2–r3 arc

– Plans that are data dependent on one another should be grouped together

We aim to achieve these goals whilst preserving the relative ordering of the calls. First, we build up a graph from the list of call clusters, with an arc between nodes that have a data-dependence or share a remote proxy. We then process the delayed-plan list in order, cluster-by-cluster.

We start by checking if there is an arc from the current cluster to its immediate successor. If there is, then we append it to the current plan. If there is not, then we check for an arc between the parent of the current cluster and the successor, appending to the parent plan if there is. We repeat the process until the check either succeeds, or there are no more parents left to check. At that point, the process is repeated with the successor cluster as the current plan. This process repeats until we have processed all the clusters.

When a remote plan B is appended to a remote plan A, a check is first made as to whether plan A is a call cluster. If it is, then a new plan is created and plans A and B inserted into it, in that order, as children, taking the place of the original plan A. If not, then B is inserted as the youngest sibling of A (i.e. B will be executed after anything already in A will be). The overall effect is that the plans form a multi-rooted tree structure, with call clusters appearing at the leaves. Plans that contain other plans are always sent to the handler of the oldest (i.e. first to be executed) sub-plan.

The algorithm currently gives equal priority to arcs due to co-location and those due to data-dependencies. It is possible to prioritise one type of arc by processing all instances of that type first when traversing through the plan hierarchy, followed by the other type.

We illustrate the process in Figure 3 using the previous example, assuming that *r1* and *r2* are targeted at the same proxy server.

4.3 Plan Caching

These optimisations incur a substantial overhead due to factors such as:

- Overhead of the Veneer runtime
- Maintenance of dependence information for delayed calls
- Pre-processing for server-forwarding
- Transmission of remote plans and metadata

We can reduce some overhead by caching plans on both server and client sides. Instead of building up remote plans by delaying calls as we encounter them, we replace the remote calls with the remote plans built up by delaying those calls previously. When the executor encounters these, it can simply place it directly onto the remote plan queue with minimal overhead.

We can only do this for adjacent clusters of remote calls rather than the merged remote plans because the pattern of remote calls might not occur next time. For example, consider Figure 4. During the first iteration, *r.f*, *r.g* and *r.h* will be aggregated, but it would not be valid to replace *r.f* with the aggregated call because the next iteration would result in *r.f*, *r.g* and *r.i* being aggregated. However, it is safe to replace *r.f* and *r.g* with the aggregate since these always occur together.

We can also take advantage of the fact that the server has seen the plan before to implement a form of data compression. The server can keep a cached copy of the plans that it receives, returning an identifier associated with the cached plan to the client. The client from that point can simply use the identifier to refer to the plan, rather than sending the entire plan every time.

```
for (int i = 0; i < 1000; i++) {
    r.f();
    r.g();
    if (i % 2 == 0)
        x = r.h();
    else
        x = r.i();
    System.out.println(x);
}
```

Fig. 4. Example of a loop that results in a different remote plan on every iteration

Client-side Implementation On the client side, we maintain a list of newly constructed call clusters. After the plans are executed, the clusters are incorporated into the method plan, such that for each cluster, all paths leading to the first call in the cluster are re-routed to the cluster, and the successor of the cluster set to the successor of the last call in the cluster. The embedded remote clusters are delayed similarly to remote calls when encountered, though without the processing required to construct the plan.

After a plan is executed, a list of cache IDs is returned by the server proxy. Cache IDs associated with call clusters are assigned directly to the embedded remote clusters. The cache IDs belonging to compound plans (plans consisting of clusters and other compound plans) are stored in a global cache, which associates a *cache pattern* with a cache ID. The cache pattern is generated by traversing the children plans of the current plan pre-order, adding the cache ID of the plans encountered as we progress.

The cache IDs for all plans are stored as a hash-map from remote server to the cache ID for that server. In all plans, we retain a handle to the last remote server used and the cache ID associated with that server. If the plan is invoked again on the same server, we can re-use the cache ID and avoid a hash-map lookup.

When the plans have been grouped and are about to be sent to the server, we attempt to send cache IDs in preference to the entire plan whenever possible using the following algorithm, starting at the root of the tree:

- If the plan is an embedded cluster, we use the associated cache ID from the embedded cluster directly.
  - If the cache ID is found, then that is used in place of the plan
  - If there is no cache ID, then we must send the entire plan
- If the plan is compound, we:
  1. Compute the cache pattern of the plan
  2. Lookup the cache ID in the global cache

  - If a cache ID is found, then it is used in place of the plan
  - If no cache ID is found, then we:
    1. Repeat the algorithm for each child of the plan
    2. If there is a cache ID for the child, then use that in place of the child plan

Server-side Implementation On the server side, the remote proxy maintains a cache of encountered plans, indexed by an integer identifier. If a remote plan containing uncached entities is executed, we cache the uncached items and return an array of cache IDs for the overall plan. Since the remote plan forms a tree structure known by both the server and client during the call, cache IDs are returned to the client as a flat array of integers by performing a pre-order traversal of the remote plan, returning the cache IDs as the nodes are encountered. The client uses this information to allocate the correct IDs to the correct clusters.

5 Maintaining Semantics

The optimisations may have changed some of the application semantics due to the difference between executing calls remotely and locally. In this section, we identify and suggest solutions to some of the problems that arise.

5.1 Differences between Local and Remote Calls

A local call and a remote call differ in the way that they pass objects as parameters. Local calls receive their parameters by reference, whereas remote calls receive them by copy. Consider the following code fragment, where `r` is a remote object:

```
a = r.f(x);
b = r.g(x);
```

Since the arguments to the call are marshalled, using local reference semantics, this would be equivalent to:

```
x' = x.clone();
a = r.f(x');
x'' = x.clone();
b = r.g(x'');
```

Note that whatever *f* does to x' is not propagated to x or x'', and similarly the effects of *g* on x'' are not propagated to x. However, by aggregating calls, the original code is transformed to the equivalent of:

```
x' = x.clone();
a = r.f(x');
b = r.g(x');
```

Now, although the effects of *f* and *g* on x' still do not affect x, the effect of *f* on x' will affect the functioning of *g*. It is therefore only safe to aggregate the two calls without copying the parameter if we can be sure that *f* does not change the value of its parameter.

An additional complication is the fact that marshalling preserves sharing between objects. For example, consider the following code:

```
x.a = y;
r.f(x, y);
r.g(x, y);
```

If we denote the arguments received by *f* as x' and y', and those received by *g* as x'' and y'', then under conventional RMI, the following properties should hold:

$$x' \neq x'' \tag{1}$$
$$y' \neq y'' \tag{2}$$
$$x'.a \neq x''.a \tag{3}$$
$$x'.a = y' \tag{4}$$
$$x''.a = y'' \tag{5}$$

This rules out copying the arguments separately, since the sharing relationship denoted by equations 4 and 5 would be broken.

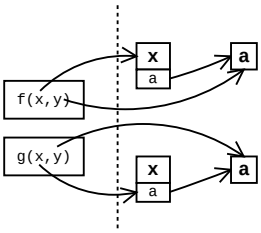


Fig. 5. Effect of sharing under object marshalling — this sharing structure cannot be maintained by copying the parameters one at a time

Copying Using Serialisation An easy way to properly copy parameters to a method call is for the server to construct an array containing the variables needed for the next call, serialise it, immediately deserialise the byte-stream into a new array, and supply the new array to the call.

Although this technique also incurs an extra cycle of serialisation and deserialisation, it is still somewhat more efficient than the simpler technique of using RMI calls locally on the server side since it avoids the overhead incurred by going through the stub and skeleton.

Avoiding Argument Copying An argument to a remote method call need not be copied if any of the following are true:

- The argument is immutable
- All objects reachable via the argument are dead after the call
- The method is guaranteed not to modify the argument

We have currently implemented some simple checks for a subset of the first two conditions. We specifically check for common object types that are known to be immutable, such as instances of `java.lang.String`.

We also introduce the notion of 'flat-types', which are types that do not contain any references. These include common types such as arrays of primitive types such as `int`. If only flat-types are used for the current and subsequent calls, then if an argument is dead and is not aliased by any other argument (which can be done simply by checking if any of the other arguments are equal to it), then we can safely avoid copying the argument.

5.2 Call-backs

When using Java RMI, it is perfectly possible for a client to act as a remote server, and vice-versa. This creates the possibility for a call-back mechanism, where a call by the client to the server will result in the server calling the client. This can create consistency problems when delaying calls.

Consider a scenario where the server `s` has managed to obtain a stub to a client `c` that also acts as a server (see Figure 6). When `c` calls `s.f(x)`, `s` makes use of the stub

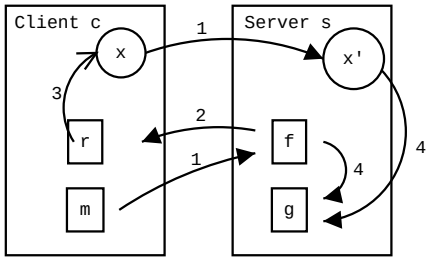


Fig. 6. The callback problem

to call the method `c.r`, which has the effect of modifying the object referenced locally on `c` by `x`. Since RMI calls are synchronous, if `g(x)` is subsequently called, the value of `x` should have been changed.

This causes a problem when aggregating calls, since the value of `x` that is sent to the server and subsequently operated on by `g` will be that of the unchanged object. However, since the client `c` generally does not know how the server `s` is implemented, it cannot tell in advance if `s` will modify `x` via `f` or not. If we ignore the problem, then `g` will end up using the old value of `x`.

A Possible Loophole It could be argued that we could simply ignore the problem due to the Java memory model in the absence of explicit synchronisation. In the Java language specification [8], the example in 7 is given:

```
class Simple {
    int a = 1, b = 2;

    void to() { a = 3; b = 4; }
    void fro() {
        System.out.println("a=" + a +
                   ", b=" + b);
    }
}
```

Fig. 7. Example to illustrate behaviour of threads accessing shared memory in the absence of synchronisation (from Java language specification)

If `to` and `fro` are called from different threads, then *a* may equal 1 or 3 and *b* may equal 2 or 4 independently. This is true even if `fro` executes after `to` has finished, since there is no obligation for `to` to write its changes back into main memory immediately without the use of synchronisation.

Since a callback must execute in a different thread from the original caller (since the caller is blocked by the unfinished RMI call), the effects of the callback might not

be immediately noticeable by the caller, in theory. In practice this does not happen due to the implementation of RMI flushing the updates to main memory, but the RMI specification [12] itself does not mandate this — in fact, it does not mention synchronisation issues at all.

Proposed Solution If we wish to ensure that the effects of callbacks are visible, then we can modify the existing protocol to do so. There are two main approaches to solving the problem — by update and by invalidation.

In the update protocol, we need the client to detect when a callback has occurred. This can be done by associating a unique session ID that is associated with the remote plan. This session ID is carried along with the plan to the remote proxy, and to any subsequent remote calls that the proxy may make. Now, if the server calls the client remotely, the client will be able to detect that it is a callback since the session ID will be known to the client. If this happens, then the client sends an updated copy of the variables associated with the session ID to the server before returning from the remote call. The server should use the fresh copy of the variables after the current call is finished.

If an invalidation protocol is used, then the server must inspect the methods being called. If a remote method may result in a callback, then the method is executed anyway, and an exception is thrown back to the client containing information regarding how far execution has progressed. The exception notifies the client of a potential callback situation, such that the client may resend the portion of the remote plan after the method that resulted in a callback, along with an up-to-date copy of the used variables.

6 Experimental Evaluations

We have tested our optimisations with two examples. The first example is a simple, synthetic benchmark to illustrate the potential of the optimisations. The second is an example of a naively written program found in the wild that may benefit from our optimisations.

The tests were performed using the Linux version of the Sun JDK version 1.4.1_01, across a Fast Ethernet network (ping time is 0.1 ms, measured bandwidth is 10.03 MB/s) and over the Internet via a slow ADSL connection (ping time is 98 ms, measured bandwidth is 10.7 kB/s). The client machine in all tests was an Athlon XP 1800+ based PC. The server for the Ethernet test was a 650MHz Intel Pentium-III PC, whilst the server for the ADSL test was a dual-processor 700MHz Pentium-III PC.

For each test, 3 trials of 1000 iterations were performed, and the mean taken as the result.

6.1 Vector Arithmetic

We have evaluated our framework using a simple synthetic benchmark in which the server object provides a single method takes two equal-sized arrays of type `double`, adds them together, and returns the resulting array. In order to test aggregation, the client application executes a sequence of remote calls of the form:

```
tmp1 = r.add(v0, v1);
tmp2 = r.add(tmp1, v2);
result = r.add(tmp2, v3);
```

This benchmark enables us to easily observe the effect of our optimisation framework as we vary the size of the data, the number of calls aggregated and various parameters of the framework.

We have tested a baseline configuration with no aggregation occurring, and configurations containing from 2–5 aggregated calls. For each configuration, we vary the vector size from 1 to 1024 doubles, doubling the vector size at every step. We test on both the Ethernet and ADSL connections.

We show the results before and after applying the framework on the benchmark program. We have also provided results for a 'hand-optimised' version of the tests (where we provide manually aggregated methods on the server and make the client call these methods) for comparison purposes.

Results As can be seen in the results in Figures 8(a)–9(e), the optimisations generally result in an overall speedup whenever any aggregation occurs. The exceptions occur when an Ethernet connection is used, with two aggregated calls and argument size of less than 400 bytes. This is due to overhead.

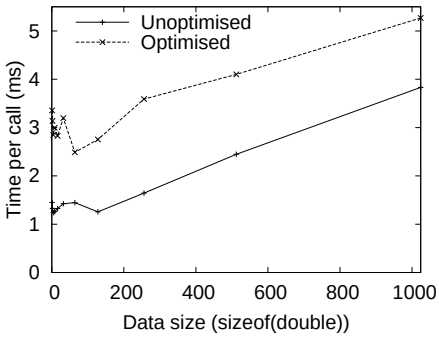
In the baseline case with no aggregation occurring, a slowdown will occur due to the same overhead being occurred but without any compensating speedup from call aggregation. This is easily observable in the Ethernet test, but is not evident in the ADSL test due to the overhead being orders of magnitude smaller compared to the communication times.

If we compare the hand-optimised versus the automatically optimised results for the tests on the Ethernet network, there is a discrepancy of about 0.5 ms per call, which is mainly due to interpretive overhead from the Veneer virtual JVM and the call-delaying/plan-building mechanism. However, this overhead remains constant, and is therefore all but invisible when operating across the Internet via ADSL, since it has much greater latencies and is subject to variations that could easily eclipse the 0.5 ms overhead.

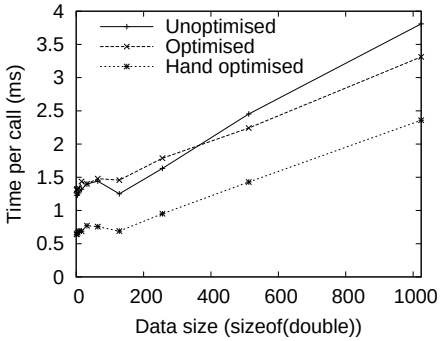
6.2 The MUD Example

The MUD (Multi-User Domain) example [6] is a more realistic example that contains call aggregation possibilities. The main candidate for optimisation occurs in the `look` method of the `MudClient` class (shown in Figure 10), which retrieves a description of the room and its contents.

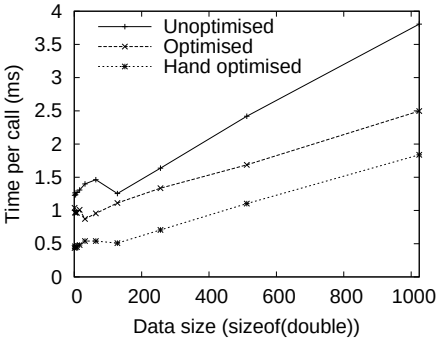
This benchmark has 7 aggregated calls with a modest payload — around 100 bytes of textual information in total. We have written a test harness that calls this routine repeatedly, recording the average time per call. Caching and server-side argument duplication have been enabled.



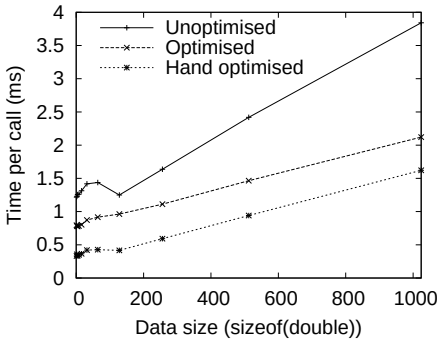
(a) No aggregation



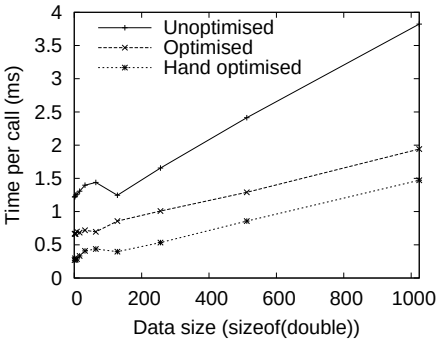
(b) Two calls aggregated



(c) Three calls aggregated

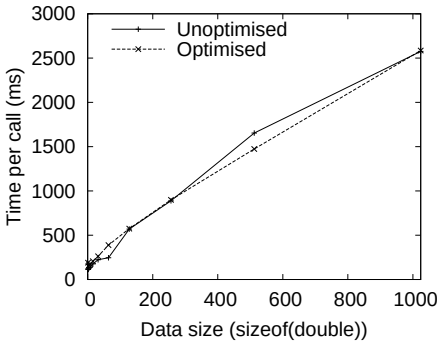


(d) Four calls aggregated

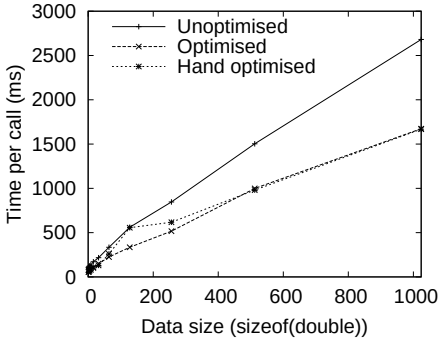


(e) Five calls aggregated

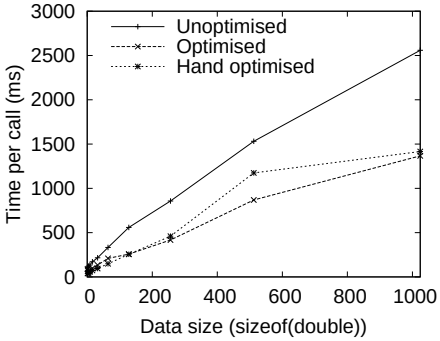
Fig. 8. Results for the vector arithmetic example running on a Fast Ethernet network with varying levels of call aggregation



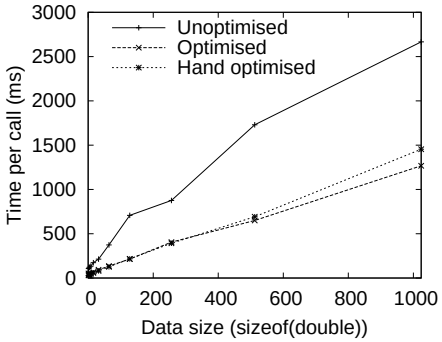
(a) No aggregation



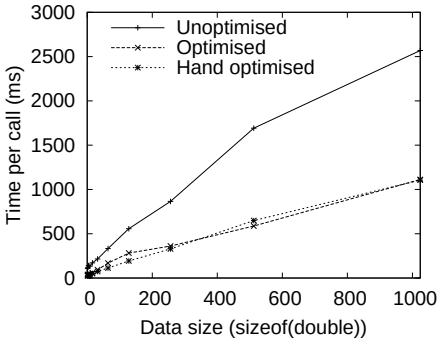
(b) Two calls aggregated



(c) Three calls aggregated



(d) Four calls aggregated



(e) Five calls aggregated

Fig. 9. Results for the vector arithmetic example running over the Internet via a slow ADSL connection with varying levels of call aggregation

```
String mudname = p.getServer().getMudName();
String placename = p.getPlaceName();
String description = p.getDescription();
Vector things = p.getThings();
Vector names = p.getNames();
Vector exits = p.getExits();
```

Fig. 10. Code for the `look` method of the MUD example

Results As can be seen in Table 1, the MUD example shows a slight slowdown when operating with an Ethernet network, but a large speedup with operating over the Internet.

Table 1. Table of results for the aggregation optimisation applied to the MUD example

Time taken to execute `look` (ms)	Without optimisation	With optimisation	Speedup
Ethernet	5.4	5.8	0.93
ADSL	759.6	164.9	4.61

The speedup is lower than what we might expect from the vectors benchmark with a similar number of aggregated calls. This is partly because there is very little variable sharing occurring between calls — the sole instance is that between `getServer` and `getMudName`, where the result of `getServer` is used as a receiver for the `getMudName` method, and is then discarded without ever reaching the client. This is in contrast to the vectors example, where each call uses the result of its predecessor.

We show a breakdown of the time taken to execute the `look` method in Table 2. As can be seen, the majority of the time in both cases is spent in client-server communication. However, on the Ethernet network, the additional overheads on the client and server side are responsible for about a third of the overall time, while the proportion of time due to overheads is insignificant by comparison when using ADSL (since the overhead remains constant while the communication times have increased). If we could minimise the overheads, then we could achieve as much as a 50% speedup when operating on an Ethernet network.

7 Future Work

Some ideas we have for enhancing the RMI optimisation further are:

– By aggregating calls, we effectively build up knowledge regarding a small portion of the client. This knowledge may enable one to perform some inter-procedural optimisations that are valid for that sequence of calls only by inlining the calls on the server side. The caching facility could serve to cache the optimised code along with the plan.

Table 2. Table showing a percentage breakdown of the time spent executing 1000 iterations of the `look` method in the MUD example

Factor	Ethernet (%)	ADSL (%)
Remote methods	0.62	0.06
Uncached RMI communication	0.78	0.35
Cached RMI communication	60.51	97.92
Client-side overhead	20.60	0.91
Server-side overhead	15.21	0.61
Argument copying overhead	2.29	0.15

- As mentioned in Section 4.1, the mechanism to detect data-dependencies triggers too easily. We intend to strengthen this with the aid of escape analysis [17], such that copying the return value of RMI calls into other data structures does not trigger a force unless that structure is visible from outside the current thread of execution.
- At present, loops are effectively unrolled as a remote plan is built up. It may be possible to export the entire loop structure to the server in order to decrease the size of the remote plan.
- Instead of considering simple 'flat-types' to decide when to avoid copying arguments, we can extend the ideas to fully-fledged balloon-types [1] to allow an arbitrary level of type-nesting, provided there are no external references.

8 Conclusion

This paper presents an attempt to extend the scope of run-time optimisation to distributed systems. Conventional optimising compilers, and optimising virtual machines, focus on each node in a system individually. This work explores optimisations which span the nodes of a distributed system. This raises many issues — including security, the potential for failure, and run-time binding of clients to servers.

We have presented a prototype tool which optimises Java RMI applications. The tool is based on a powerful framework, essentially a 'virtual' JVM, which allows the run-time system to re-order blocks of application code subject to data dependence meta-data generated by static analysis. We use this to implement two optimisations of RMI applications: call aggregation, and call forwarding. These, in turn, lead to further optimisations, such as eliminating data transfer across the network for data passed between aggregated calls.

We present performance results for simple examples which demonstrate the performance potential for these optimisations. We also show preliminary results for a more substantial application, which demonstrate that optimisation opportunities do arise in real systems.

Our prototype implementation is based on a very powerful experimental framework, and this incurs some run-time overheads which we hope to reduce in time. There is enormous scope for more powerful analysis and more ambitious optimisations.

9 Acknowledgements

This project was funded by a studentship and grant from the Engineering and Physical Sciences Research Council (GR/R 15566).

References

[1] Paulo Sérgio Almeida. Balloon types: Controlling sharing of state in data types. In *Proceedings of ECOOP '97*, volume 1241 of *Lecture Notes in Computer Science*, pages 32–59. Springer, June 1997.

[2] Phillip Bogle and Barbara Liskov. Reducing cross domain call overhead using batched futures. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 341–354, Portland OR (USA), 1994.

[3] R. Christ, S. L. Halter, K. Lynne, S. Meizer, S. J. Munroe, and M. Pasch. SanFrancisco performance: A case study in performance of large-scale Java applications. *IBM Systems Journal*, 39(1), 2000.

[4] Markus Dahm. Byte code engineering library manual. Available from http://jakarta.apache.org/bcel/manual.html.

[5] John Eberhard and Anand Tripathi. Efficient object caching for distributed Java RMI applications. In *Middleware 2001, Proceedings*, volume 2218 of *Lecture Notes in Computer Science*, pages 15–35. Springer, November 2001.

[6] David Flanagan. *Java Examples in a Nutshell*. O'Reilly UK, 2000.

[7] Aniruddha Gokhale and Douglas C. Schmidt. Principles for optimizing CORBA internet inter-ORB protocol performance. In *31th Hawaii International Conference on System Sciences*, January 1998.

[8] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification — Second Edition*. Addison-Wesley, 2000.

[9] Object Management Group. The Common Object Request Broker: Architecture and specification v2.4.2, February 2001.

[10] Vijaykumar Krishnaswamy, Dan Walther, Sumeer Bhola, Ethendranath Bommaiah, George Riley, Brad Topol, and Mustaque Ahamad. Efficient implementations of Java remote method invocation (RMI). In *Proc. of the 4th USENIX Conference on ObjectOriented Technologies and Systems (COOTS'98), 1998.*, pages 19–36, 1998.

[11] Barbara Liskov and Liuba Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proceedings of the SIGPLAN'88 conference on Programming Language Design and Implementation*, pages 260–267, 1988.

[12] Sun Microsystems. RMI specification, available at http://java.sun.com/products/jdk/rmi/.

[13] Christian Nester, Michael Phillippsen, and Bernhard Haumacher. A more efficient RMI for Java. In *ACM 1999 Java Grande Conference*, pages 152–159, June 1999.

[14] Michael Philippsen, Bernhard Haumacher, and Christian Nester. More efficient serialization and RMI for Java. *Concurrency: Practice and Experience*, 12(7):495–518, 2000.

[15] R. Raje, J. William, and M. Boyles. An Asynchronous Remote Method Invocation (ARMI) mechanism for Java. *Concurrency: Practice and Experience*, November 1997.

[16] Raja Vallee-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a Java bytecode optimization framework. In *Proceedings of CASCON '99*, pages 125–135, 1999.

[17] John Whaley and Martin Rinard. Compositional pointer and escape analysis for Java programs. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, volume 34 of *ACM SIGPLAN Notices*, pages 187–206, November 1999.